A Low Complexity Block Turbo Decoder Architecture

C. Vanstraceele, B. Geller, J.M. Brossier, J.P. Barbot

Abstract

We present a low-complexity architecture designed for the decoding of block turbo codes. In particular we simplify the implementation of Pyndiah's algorithm by not memorizing any of the concurrent codewords generated by the Chase search.

Index Terms

Block codes, Iterative decoding, Low-complexity design, Turbo codes.

Affiliations of the authors:

C. Vanstraceele, B. Geller and J.P. Barbot are with Laboratory SATIE, ENS Cachan, 61 av. du Président Wilson, 94235 Cachan cedex, France.

J.M. Brossier is with Laboratory LIS, UMR 5083, INP Grenoble, 961 av. de la Houille Blanche, B.P. 46 38402 Saint Martin d'Hères cedex, France.

Corresponding author: B. Geller, Laboratoire SATIE, ENS Cachan, 61 av. du Président Wilson, 94235 Cachan cedex, France. Tel:(33)680473174. Fax:(33)147402199.e-mail: benoit.geller@satie.ens-cachan.fr

A Low Complexity Block Turbo Decoder Architecture

I. Introduction

Forward-Error Correction (FEC) is used in many digital communication systems to provide reliability and coding gains over the overall power budget of the link. Convolutional turbo-codes allow performance close to Shannon's theoretical limit. Block Turbo Codes (BTCs), *i.e.* product block codes with iterative column and row decodings, also achieve near-capacity decoding with high coding rates [1].

Pyndiah [2] has devised a low-complexity block turbo decoding algorithm. Each iteration m of this algorithm can be summarized as follows:

- For each row (or column) of an $n \times n$ product word, perform a Chase search [3] in order to have a list of codewords and a decided code word $\mathbf{d} = (d_0, \ldots, d_j, \ldots, d_{n-1})$ with $d_j \in \{-1, +1\}$.
- Evaluate a vector of decision reliabilities $\mathbf{\Lambda} = (\Lambda(d_0), \ldots, \Lambda(d_j), \ldots, \Lambda(d_{n-1}))$ in terms of normalized log-likelihood ratio (LLR); for each position j ($j < n$), the reliability of each decision is:
$$\Lambda(d_j) \approx \frac{\left|\mathbf{r} - \mathbf{c}^{-1(j)}\right|^2 - \left|\mathbf{r} - \mathbf{c}^{+1(j)}\right|^2}{4}, \tag{1}$$
where $\mathbf{c}^{-1(j)}$ (respectively $\mathbf{c}^{+1(j)}$) is the codeword in the Chase list at minimum Euclidean distance of the updated received codeword $\mathbf{r} = (r_0, \ldots, r_j, \ldots, r_{n-1})$, such that $c^{-1(j)} = -1$ (respectively $c^{+1(j)} = +1$). One of these two codewords is equal to $\mathbf{d}$ and the other is then referred as $\mathbf{c_d}$.
- Obtain for each position j the extrinsic information w_j:
$$w_j = \begin{cases} \Lambda(d_j) - r_j \text{ if a competing } \mathbf{c_d} \text{ exists,} \\ \beta(m) d_j \text{ if no competing word exists.} \end{cases} \tag{2}$$
- Once the extrinsic information has been evaluated, update the input to the next Soft-Input/Soft-Output (SISO) decoding stage with:

$$r'_j = r_j + \alpha(m) w_j \tag{3}$$

where $\alpha(m)$ is a weight factor increasing along the convergence process.

There were several refinements to Pyndiah's original work. For instance, [4] (respectively [5]) proposes to update the algorithm without the need of using any of the coefficients $\beta(m)$ in equation (2) (respectively $\alpha(m)$ in equation (3)). [6] considers parallel rows and columns decoding of the product code in order to half the latency of the decoder. [7] presents a low-complexity Chase decoder.

The purpose of this letter is to show that it is absolutely unnecessary to put any of the Chase codewords into memory and that at each iteration, one can readily update the output of the SISO decoder.

II. Simplification of the implementation

A. Principle

We consider the decoding process of a given updated received (line or column) vector $\mathbf{r}$ of length n. Let $p(\mathbf{c}) = \sum_{j=0}^{n-1} r_j c_j$ be the ordinary scalar product of $\mathbf{r}$ with a codeword $\mathbf{c}$. A codeword is at minimum distance of $\mathbf{r}$ if, and only if, its scalar product is maximum. Equation (1) can be written as the difference between two scalar products:

$$\Lambda(d_j) = \frac{1}{2}\left[p\left(\mathbf{c}^{+1(j)}\right) - p\left(\mathbf{c}^{-1(j)}\right)\right] \tag{4}$$

This illustrates that instead of memorizing codewords to evaluate the reliabilities, it is sufficient to memorize scalar products [8], [9]. One then observes that one such memorized scalar product can be used several times during the updating of a vector of reliabilities $\mathbf{\Lambda}$ if the corresponding code word is at minimum Euclidean distance for several different coordinates. More precisely, a given scalar product will be used every time that the corresponding code word is one of the possibly two minimum distance competing codewords. For instance, for every coordinate, the scalar product $p(\mathbf{d})$ of the decided codeword $\mathbf{d}$ will necessarily be one of the two competing scalar product of equation (4) as the decoded word $\mathbf{d}$ is at minimum Euclidean distance of $\mathbf{r}$.

We thus adopt the following strategy requiring only two shift registers of n real values.

$\mathbf{s}^+ = (s_0^+, ..., s_j^+, ..., s_{n-1}^+)$ designates a shift register of scalar products $p(\mathbf{c}^{+1(j)})$.

$\mathbf{s}^- = (s_0^-, ..., s_j^-, ..., s_{n-1}^-)$ designates a shift register of scalar products $p(\mathbf{c}^{-1(j)})$.

Initialization :

$\mathbf{s}^+$ and $\mathbf{s}^-$are both initialized at the minimum possible values ($-\infty$).

First step :

Let $\mathbf{c}_1 = (c_{1,0}, \ldots, c_{1,j}, \ldots, c_{1,n-1})$ be the first Chase decoded codeword with scalar product $p(\mathbf{c}_1)$,
$J_1^+ = \{j \in \{0, 1, \cdots, n-1\} | c_{1,j} = +1\}$,
$J_1^- = \{j \in \{0, 1, \cdots, n-1\} | c_{1,j} = -1\}$.
For $j \in J_1^+$, $s_j^+ = P[\mathbf{c}_1]$.
For $j \in J_1^-$, $s_j^- = P[\mathbf{c}_1]$.

Following steps :

For each different Chase decoded $\mathbf{c}_k = (c_{k,0}, \ldots, c_{k,j}, \ldots, c_{k,n-1})$ with scalar product $P(\mathbf{c}_k)$,
$J_k^+ = \{j \in \{0, 1, \cdots, n-1\} | c_{k,j} = +1\}$,
$J_k^- = \{j \in \{0, 1, \cdots, n-1\} | c_{k,j} = -1\}$.
For $j \in J_k^+$, if $p(\mathbf{c}_k) > s_j^+$ then $s_j^+ = p(\mathbf{c}_k)$.
For $j \in J_k^-$, if $p(\mathbf{c}_k) > s_j^-$ then $s_j^- = p(\mathbf{c}_k)$.

Result :

$\mathbf{\Lambda} = (\Lambda(d_0), \ldots, \Lambda(d_j), \ldots, \Lambda(d_{n-1})) = \frac{1}{2}(\mathbf{s}^+ - \mathbf{s}^-)$ (The indexes where there is no competing codeword $\mathbf{c_d}$ are simply β positions - see equation (2) or reference [4]).

B. Example

We now illustrate the previous procedure with a simple example. Let $\mathbf{c}_i$, be the four codewords generated by a Chase search with their scalar product $p(\mathbf{c}_i)$ ($i \in \{1, 2, 3, 4\}$) evaluated by the decoder.

First step :

$$\mathbf{c}_1 = (-1,+1,-1,+1,+1,-1,-1,+1), p(\mathbf{c}_1) = 1.6$$
$$\mathbf{s}^+ = (-\infty, 1.6, -\infty, 1.6, 1.6, -\infty, -\infty, 1.6)$$
$$\mathbf{s}^- = (1.6, -\infty, 1.6, -\infty, 1.6, 1.6, -\infty).$$

Second step :

$$\mathbf{c}_2 = (+1,-1,+1,-1,+1,+1,-1,-1), p(\mathbf{c}_2) = 2.5$$
$$\mathbf{s}^+ = (2.5, 1.6, 2.5, 1.6, 2.5, 2.5, -\infty, 1.6)$$
$$\mathbf{s}^- = (1.6, 2.5, 1.6, 2.5, -\infty, 1.6, 2.5, 2.5).$$

Third step :

$$\mathbf{c}_3 = (+1,+1,+1,+1,-1,+1,-1,+1), p(\mathbf{c}_3) = 6.4$$
$$\mathbf{s}^+ = (6.4, 6.4, 6.4, 6.4, 2.5, 6.4, -\infty, 6.4)$$
$$\mathbf{s}^- = (1.6, 2.5, 1.6, 2.5, 6.4, 1.6, 6.4, 2.5).$$

Fourth step :

$$\mathbf{c}_4 = (+1,-1,+1,+1,-1,-1,+1,-1), p(\mathbf{c}_4) = 2.4$$
$$\mathbf{s}^+ = (6.4, 6.4, 6.4, 6.4, 2.5, 6.4, 2.4, 6.4)$$
$$\mathbf{s}^- = (1.6, 2.5, 1.6, 2.5, 6.4, 2.4, 6.4, 2.5).$$

Result :

$\mathbf{\Lambda} = \frac{1}{2}(\mathbf{s}^+ - \mathbf{s}^-) = (+2.4, +1.95, +2.4, +1.95, -1.95, +2.0, -2.0, +1.95)$.

We notice that the maximum scalar product vector $\mathbf{c}_3 = \mathbf{d}$ is such that

$\mathbf{c}_3 = (\text{sign}[\Lambda(d_0)], \ldots, \text{sign}[\Lambda(d_j)], \ldots, \text{sign}[\Lambda(d_{n-1})])$.

C. Further discussion

The previous procedure simplifies the whole architecture of a BTC decoder. Not only there is no need to put into memory the competing codewords, but also, with the simplification of the reliabilities evaluation, many of the existing functionalities of a BTC decoder [10] disappear as, for instance, the search for every bit of a relevant competing codeword. It is even unnecessary to identify which is the maximum likelihood vector $\mathbf{d}$ among the competing codewords.

One can take advantage of the reduction in architecture complexity to use more test patterns dedicated to the Chase search. Figure 1 illustrates in the case of a BCH$(128, 113)^2$ product code

the Bit Error Rate (BER) improvement at the fourth iteration when one increases the number of least reliable bits used for the generation of the test patterns from $t = 4$ to $t = 6$. A gain of 0.3 dB is then observed at a BER of 10^{-5} which leads this code of coding rate $R = 0.78$ to be at 2.1 dB from the Shannon's capacity in only 4 iterations.

The proposed procedure is particularly light when used in conjunction with a Fast Chase algorithm [7]; this is because if two words $\mathbf{c}$ and $\mathbf{c}^{'}$ differ in one position k ($c_k^{'} = -c_k$), their scalar products are linked by the equality $p(\mathbf{c}^{'}) = p(\mathbf{c}^{'}) - r_k c_k^{'}$.

We now display the improvement on the complexity brought by the proposed architecture in terms of required memory. This architecture being totally independant on the way the concurrent codewords are generated, we suppose that for a given row (or line) of the product code, one has obtained a list of codewords. Most often, a Chase algorithm is chosen to generate this codewords list for both Pyndiah's original architecture and the present proposal.

Let q be the number of quantification bits and N_c be the number of generated codewords; in the sequel we display numerical results where N_c is equal to 2^p. With Pyndiah's traditional algorithm, one has to store N_c codewords with the associated Euclidean distance. One thus has to store N_b^{PY} bits where:

$$N_b^{PY} = N_c(n + q) \tag{5}$$

With the proposed algorithm, one has to store two real vectors $\mathbf{s}^+$ and $\mathbf{s}^-$; one thus needs N_b^{PR} bits where:

$$N_b^{PR} = 2qn \tag{6}$$

The following table allows to compare the required memory for typical parameters values :

	N_b^{PY}	N_b^{PR}
$n = 32$, $N_C = 16$, $q = 4$	576	256
$n = 32$, $N_C = 16$, $q = 5$	592	320
$n = 32$, $N_C = 64$, $q = 4$	2304	256

TABLE I

COMPARISON OF THE REQUIRED MEMORY

It is straightforward that when one wants to store more codewords, the required memory with the traditional Pyndiah's decoder becomes large and this becomes critical as in practice several elementary decoders are used in parallel. To circumvent this inconvenient, practical decoders just take into account a limited number of codewords. This leads to a Bit Error Rate degradation; on the contrary, the required memory of the proposed method is not affected by the number of generated codewords.

III. Conclusion

This letter presented a particularly low-complexity procedure to implement block turbo decoding and giving exactly the same results as Pyndiah's work [2]. This low-complexity allows the implementation of larger codes for which Pyndiah's algorithm gives near-capacity results. This algebraic based decoding scheme seems particularly suited for high data rate systems.

References

[1] S. Hagenauer, E. Offer, and L. Papke, "Iterative decoding of binary block and convolutional codes," *IEEE Trans. Inform. Theory*, vol. 42, no. 2, pp. 429–445, March 1996.

[2] R. Pyndiah, "Near optimum decoding of product codes : Block Turbo Codes," *IEEE Trans. Commun.*, vol. 46, no. 8, pp. 1003–1010, August 1998.

[3] D. Chase, "A class of algorithms for decoding block codes with channel measurement information," *IEEE Trans. Inform. Theory*, vol. 18, no. 1, pp. 170–182, January 1972.

[4] P. Adde and R. Pyndiah, "Recent simplifi cations and improvements in Block Turbo Codes," in *Proc. 2nd int. Symposium on Turbo Codes and related Topics*, Brest, Sept. 2000, pp. 133–136.

[5] Z. Chi, L. Song, and K. Parhi, "On the performance/complexity tradeoff in block turbo codes," *IEEE Trans. Commun.*, vol. 52, no. 2, pp. 173–175, February 2004.

[6] C. Argon and S. McLaughlin, "A parallel decoder for low latency decoding of turbo product codes," *IEEE Commun. Letters*, vol. 6, no. 2, pp. 70–72, February 2002.

[7] S. Hirst, B. Honary, and G. Markarian, "Fast Chase algorithm with an application in turbo decoding," *IEEE Trans. Commun.*, vol. 49, no. 10, pp. 1693–1699, October 2001.

[8] B. Geller, "Contribution à l'étude des systèmes de communications numériques," Habilitation à diriger des recherches, Université Paris 12, 2004.

[9] C. Vanstraceele, "Turbo-Codes et estimation paramétrique pour les communications à haut débit," Ph.D. dissertation, Ecole Normale Superieure de Cachan, France, 2005.

[10] S. Kerouedan, P. Adde, and R. Pyndiah, "How we implemented Block Turbo Codes," *Annals of telecommunications*, vol. 56, no. 7-8, pp. 447–454, 2001.

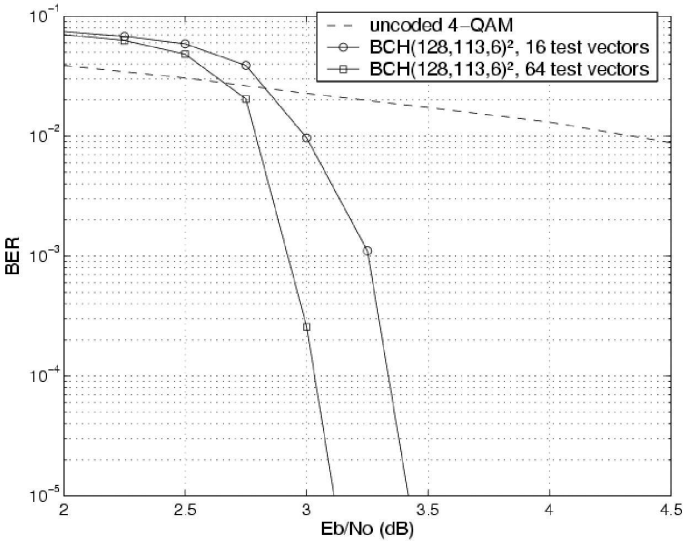

Fig. 1. Bit Error Rates at iteration $m = 4$ for different numbers of tested bits using QPSK signalling on an additive white Gaussian noise channel.